National University of Singapore
School of Computing
CS1101S: Programming Methodology (JavaScript)
Semester I, 2012/2013

Discussion Group Exercises 5

Problems:

1. Concrete Abstractions, Exercise 13.33

   The three procedures below are the constructor, mutator, and selector for a new kind of object, the widget. Describe in English how widgets behave, from the standpoint of someone using these three procedures but not knowing what is going on inside them or how the widgets are being represented. That is, your explanation shouldnt talk about vectors or vector positions at all but instead should talk about how widget insertion and retrieval relate. If some insertions and retrievals are done, how could you predict what each retrieval was going to retrieve? Once youve provided this outsiders perspective, provide a justification of it in terms of the internal behavior of the procedures. That is, explain how it is that the vector operations these procedures do result in the previously stated external behavior.

   A vector acts like a list, except that it has O(1) access time instead of O(n). vector_set(vector, position, value) changes the object at position to a new value. vector_ref(vector, position) gets the object at position.

   ```
   function make_widget() {
       var widget = make_vector(3);
       vector_set(widget, 0, "empty");
       vector_set(widget, 1, "empty");
       vector_set(widget, 2, 0);
       return widget;
   }

   function insert_into_widget(widget, value) {
       var place = vector_ref(widget, 2);
       vector_set(widget, place, value);
       vector_set(widget, 2, (place + 1) % 2);
       return "done";
   }

   function retrieve_from_widget(widget) {
       return vector_ref(widget, vector_ref(widget, 2));
   }
   ```

2. SICP, Exercise 3.1

   An accumulator is a function that is called repeatedly with a single numeric argument and accumulates its arguments into a sum. Each time it is called, it returns the currently accumulated sum. Write a function make_accumulator that generates accumulators, each maintaining an independent sum. The input to make_accumulator should specify the initial value of the sum.

```
    var A = make_accumulator(5);
    A(10);
    > 15
    A(10);
    > 25
```

3. SICP, Exercise 3.2

   In software testing applications, it is useful to be able to count the number of times a given function is called during the course of a computation. Write a function make_monitored that takes as input a function, f, that itself takes one input. The result returned by make_monitored is a third function, say mf, that keeps track of the number of times it has been called by maintaining an internal counter. If the input to mf is the string ”get_count”, then mf returns the value of the counter. If the input is the string ”reset_count”, then mf resets the counter to zero. For any other input, mf returns the result of calling f on that input and increments the counter. For instance, we could make a monitored version of the Math.sqrt function:

```
    var s = make_monitored(Math.sqrt);

    s(100);
    > 10
    s("get_count");
    > 1
    s("reset_count");
    s("get_count");
    > 0
```

4. SICP, Exercise 3.5

   Monte Carlo integration is a method of estimating definite integrals by means of Monte Carlo simulation. Consider computing the area of a region of space described by a predicate P(x, y) that is true for points (x, y) in the region and false for points not in the region. For example, the region contained within a circle of radius 3 centered at (5, 7) is described by the predicate that tests whether (x - 5)$\hat{2}$ + (y - 7)$\hat{2}$ ¡ 3$\hat{2}$. To estimate the area of the region described by such a predicate, begin by choosing a rectangle that contains the region. For example, a rectangle with diagonally opposite corners at (2, 4) and (8, 10) contains the circle above. The desired integral is the area of that portion of the rectangle that lies in the region. We can estimate the integral by picking, at random, points (x, y) that lie in the rectangle, and testing P(x, y) for each point to determine whether the point lies in the region. If we try this with many points, then the fraction of points that fall in the region should give an estimate of the proportion of the rectangle that lies in the region. Hence, multiplying this fraction by the area of the entire rectangle should produce an estimate of the integral.

   Implement Monte Carlo integration as a procedure estimate_integral that takes as arguments a predicate P, upper and lower bounds x1, x2, y1, and y2 for the rectangle, and the number of trials to perform in order to produce the estimate. Your procedure should use the same monte-carlo procedure that was used above to estimate . Use your estimate_integral to produce an estimate of by measuring the area of a unit circle.

   You will find it useful to have a procedure that returns a number chosen at random from a given range. The following random_in_interval procedure implements this.

```
function random_in_interval(from,to) {
    return Math.random()*(to-from)+from;
}
```

5. The following procedure is quite useful, although obscure:

   ```
   function mystery(x) {
       function loop(x, y) {
           if (is_empty_list(x)) {
               return y;
           } else {
               var temp = tail(x);
               set_tail(x, y);
               return loop(temp, x);
           }
       }
       return loop(x, []);
   }
   ```

   Loop uses the temporary variable temp to hold the old value of the tail of x, since the set_tail on the next line destroys the tail. Explain what mystery does in general. Suppose v is defined by var v = list(1, 2, 3, 4). Draw the box-and-pointer diagram that represents the list to which v is bound. Suppose that we now evaluate var w = mystery(v). Draw box-and-pointer diagrams that show the structures v and w after evaluating this expression. What would be printed as the values of v and w?

6. SICP, Exercises 3.16 and 3.17

   Ben Bitdiddle decides to write a procedure to count the number of pairs in any list structure. ”Its easy,” he reasons. ”The number of pairs in any structure is the number in the head plus the number in the tail plus one more to count the current pair.” So Ben writes the following function:

   ```
   function count_pairs(x) {
       if (!is_pair(x)) {
           return 0;
       } else {
           return 1 + count_pairs(head(x)) + count_pairs(tail(x));
       }
   }
   ```

   Show that this function is not correct. In particular, draw box-and-pointer diagrams representing list structures made up of exactly three pairs for which Bens procedure would return 3; return 4; return 7; never return at all.

   Devise a correct version count-pairs that returns the number of distinct pairs in any structure. (Hint: Traverse the structure, maintaining an auxiliary data structure that is used to keep track of which pairs have already been counted.)

7. **Sorting Hat**
   i) Write a function find_smallest that, given a list of integers, will return the first pair in the list whose head is the smallest integer in the whole list. It should return [] if the given list is empty. Determine the order of growth of this procedure. Sample runs:

   ```
   find_smallest([]);
   > []
   find_smallest(list(1,2,3));
   > [1, [2, [3, []]]]
   find_smallest(list(3,2,1));
   > [1, []]
   find_smallest(list(6,6,4,2,4,3,5,2,3));
   > [2, [4, [3, [5, [2, [3, []]]]]]]
   ```

ii) Write a new function find_pair_before_smallest by modifying or making use of find_smallest to return a list beginning with the element before the smallest integer in the list. The function should return [] if the smallest integer is already at the front of the list. Sample runs:

```
find_pair_before_smallest(list(6, 6, 4, 2, 4, 3, 5, 2, 3));
> [4, [2, [4, [3, [5, [2, [3, []]]]]]]]
find_pair_before_smallest(list(1, 2, 3));
> []
```

iii) Write a function shift_smallest that, given a list of integers, will modify the list such that the first smallest integer in the list is shifted to the front of the list and returning that list. It should not modify the order of the rest of the elements of the list at all. Hint: You will need to use set-head or set-tail in this question. Sample runs:

```
var a = shift_smallest(list(1,2,3));
a
> [1, [2, [3, []]]]
var b = shift_smallest(list(6,6,4,2,4,3,5,2,3));
b
> [2, [6, [6, [4, [4, [3, [5, [2, [3, []]]]]]]]]]
var a = list(3,2,1);
var b = shift_smallest(a);
a
> [3, [2, []]]
b
> [1, [3, [2, []]]]
```

iv) Armed with the function shift_smallest, we are now able to write a function that sorts a list of integers from smallest to biggest. The function sort_list accepts a list of integers as its argument, and modifies the list such that the members are sorted from smallest to biggest. Determine the order of growth if this function.

v) A comparator is a general term for a function that takes two parameters (not necessarily numbers) and returns true if the first parameter is strictly smaller than the second parameter. In this sense, a comparator determines the ordering of the two parameters. Here is an example of a comparator:

```
function person_compare(p1, p2) {
    return p1("get_name") < p2("get_name");
}
```

Write a function sort_list_with_comparator that accepts a list of things and a comparator that is able to compare any two members of the given list. The function should sort the list with respect to the ordering specified by the given comparator.

vi) Is the two sorting procedures you implemented above stable? Do you know what is the name of the sorting algorithm we just implemented?